Introduction to Spring and Spring MVC

20

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Spring Framework.
- Inversion of Control and Dependency Injection.
- Dependency injection variants.
- Bean Scopes such as Singleton and Prototype.
- Spring MVC.
- Role of controller and how MVC works.
- Role of Dispatcher Servlet.
- Concept of Dependency Injection and Inversion of Control.
- Maven and how to use it.
- How to connect database to perform CRUD operations.
- How to design a view to display data processed by Model through Controller.

20.1 | Spring Framework

In the concept of web application at a higher level, we have a presentation layer, a business layer, and a database layer. The days before Spring was invented, most of the applications were developed using the concept called *procedural programming* in which a specific task is done by calling the required libraries by the logic code. In other words, a task is achieved by performing a series of computational steps in a specific order. Spring is built on concepts called *inversion of control* and *dependency injection* (IoC and DI in short, respectively). It allows building applications from Plain Old Java Objects (POJOs) and apply other required services in a non-invasive manner to achieve the desired functionality.

QUICK CHALLENGE Mention all the modules of the Spring framework.

Let us explore these topics in detail to understand the inner working of Spring.

20.1.1 Inversion of Control

As we have seen in the traditional programming model, the logic code makes calls to the required libraries to perform a series of tasks. IoC does the exact opposite, where the control remains with the framework which executes the logic code. This promotes loose coupling, which means the objects of the functional classes do not depend on the other required objects' concrete implementation.

IoC provides various benefits such as modularity, loose coupling, etc., where the functionalities of the program get divided into various areas. This allows loose coupling where the components are not dependent on each other's implementation. Hence, it offers flexibility to modify the implementation of these objects without affecting the other parts of the application. This also enables testing teams to test the functionalities in isolation and later test the dependable functionality by mocking the object dependencies.

IoC can be achieved with the help of DI, which we will explore now.

20.1.2 Dependency Injection

In a typical application, there is a need to work with a lot of individual objects in order to develop a required functionality. For example, for a school registration system, we will need a student object, teacher object, classroom object, subject object, etc. These objects should be accessible in a business logic class such as Registration.java, which will enable the user to register for courses. In traditional development, you will need to initialize all these objects in order to use them. In other words, you will create these objects by yourself and need to manage the life cycle of each object for the optimum use of resources such as memory. This is where DI comes handy. DI is a structural design pattern that eliminates the need for us to initialize these required objects and manage the life cycle by ourselves. It solves this problem by injecting the required object to the constructor or setter by passing as a parameter. Figure 20.1 shows the use of DI. This DI functionality is implemented in a library called *inversion of control* (IoC) container. This IoC container is responsible for injecting the dependent objects when the bean creation is taking place. Since this operation is inverse of the traditional approach, it is called inversion of control.

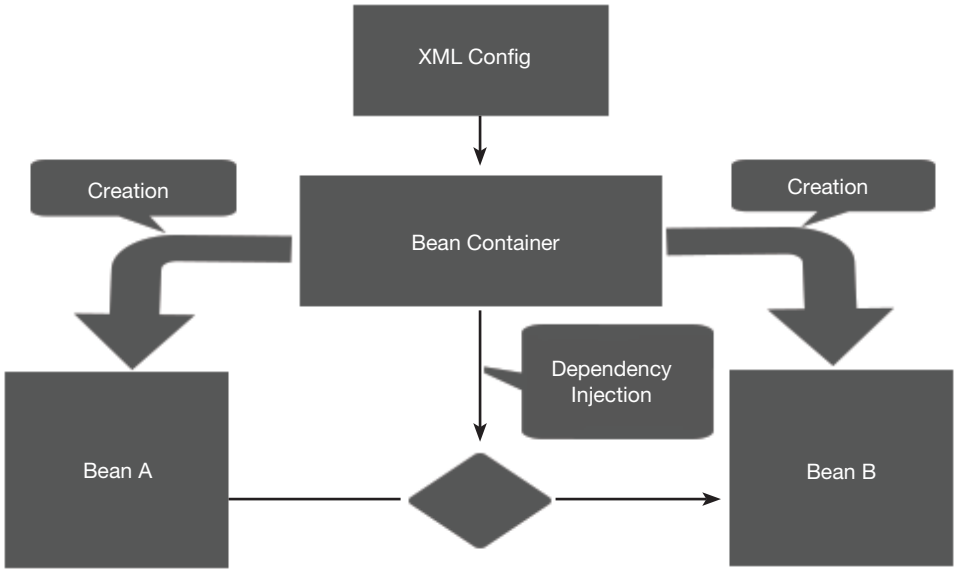

Figure 20.1 Representation of use of dependency injection.

Let us understand this better with an example. We will first take a look at the traditional approach, in which we will create an object dependency.

```
package java11.fundamentals.chapter20;
public class ObjectDependencyTraditional {
    private Product product;
    public ObjectDependencyTraditional() {
        product = new Product("My Awesome Product");
    }
}
class Product{
    private String name;

    public Product(String name) {
        this.name = name;
    }
}
```

In the above example, we have to instantiate the dependent object "product" within the ObjectDependencyTraditional class. Now, let us take a look at the DI approach.

```
package java11.fundamentals.chapter20;
public class ObjectDependencyWithDI {
    private ProductDI product;
    public ObjectDependencyWithDI(ProductDI product) {
        this.product = product;
    }
}
class ProductDI{
    private String name;

    public ProductDI(String name) {
        this.name = name;
    }
}
```

In the above example, IoC will inject the object of ProductDI while creating ObjectDependencyWithDI bean.

There are multiple ways you could use DI. Before exploring these, let us first understand how the IoC container works. IoC is mainly responsible for instantiating, configuring, assembling objects, and managing their life cycles. These objects are known as *beans*. Spring provides ApplicationContext interface to implement DI mechanism. This interface is a subinterface of BeanFactory interface which provides advanced configuration method to manage all types of objects. We can consider BeanFactory as the central registry which holds the configuration of all the application components. ApplicationContext extends it to add more enterprise-specific functionality. ApplicationContext interface has several implementations for various purposes; for standalone application it provides FileSystemXmlApplicationContext, for web applications it has WebApplicationContext and ClassPathXmlApplicationContext to provide a configuration for the context definition file.

Let us see how to instantiate a container to use to get the managed objects instances.

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("MyApplicationContext.xml");

In the above example, we are using ClassPathXmlApplicationContext to get objects configurations from an XML file that is located on the class path, which is then used by the container to assemble beans at runtime.

Before we deep dive into variants of DIs, let us first explore the concept called *bean scope*. This is an important concept as it will help improve the application performance.

How do we perform dependency injection in Spring framework?

20.1.2.1 Bean Scope

In the above section, we have seen how IoC container using DI creates beans that are needed by the application. Although the IoC container creates these required beans for us, Spring provides a mechanism to have more granular control on the creation process by specifying the bean scope. The Spring framework provides five scopes. These are singleton, prototype, request, session, and global-session. Let us explore these scopes in detail:

1. **singleton:** The framework first looks for a cached instance of the bean and if it cannot find one, it creates a new one.
2. **prototype:** The framework will create a new bean instance on every request. In order to use the other three scopes listed below, you need to use a web-aware ApplicationContenxt.
3. **request:** This is similar to the prototype scope but only for web applications, where a new instance is created on every HTTP request.
4. **session:** As the name suggests, this scope is limited to every HTTP session. A new instance is available per session.
5. **global-session:** This scope is useful for portlet applications where a new instance is created for every global session.

Here is how you define the scope:

```
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="java11.fundamentals.
chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample" scope="singleton" >
    <constructor-arg type="java.lang.Boolean" value="true"/>
    <constructor-arg type="java.lang.String" value="Mayur Ramgir"/>
</bean>
```

In the above example, *<bean>* definition provides an attribute called *scope* where we can specify the type of scope we need. If no "scope" attribute is provided, Spring considers the bean as "singleton". In other words, singleton is the default bean scope.

There are multiple ways we could use DI in our application. In subsection 20.1.2.3, we will explore these ways in detail. However, before deep diving into this concept, we must first understand the concept called Autowiring. This is a mechanism used by Spring container to inject dependencies to the class.

20.1.2.2 Autowiring Dependencies

Spring container uses autowiring to automatically resolve dependencies. There are four modes by which we can autowire a bean in XML configuration.

1. **Default mode:** In this mode, no autowiring is used and we have to manually define the dependencies.
2. **byName mode:** In this mode, the autowiring is done using the name of the property. This name is used by Spring to look for a bean to inject.
3. **byType mode:** In this mode, the type of the property is used by Spring to look for a bean. If more than one beans are found, Spring throws an exception.
4. **Constructor mode:** In this mode, constructor of the class is used to autowire the dependencies. Spring will look for beans defined in the constructor arguments.

Are singleton beans threadsafe?

20.1.2.3 Variants of Dependency Injection

There are mainly three types of variants of DI, which can be used to set the dependencies for the class. These are constructor-based, setter-based, and field-based DI. Constructor-based DI is the most recommended way of injecting the required dependencies. It has many advantages over the other two methods. Setter-based injection is the next recommended way. It is highly discouraged to use field-based DI due to several disadvantages. Let us explore these in more detail to understand the reasons for these recommendations.

20.1.2.3.1 Constructor-Based Dependency Injection

This method is useful to inject the required dependencies via constructor arguments. Let us see the following example that shows a class in which dependencies are supplied via constructor arguments. Please note that this is a simple POJO class which does not have any dependencies on any of Spring container's interfaces, classes, or annotations.

```
package java11.fundamentals.chapter20;
public class ConstructorBasedDependencyInjectionExample {
    private Tyre tyre;
    private HeadLights headlight;
    public ConstructorBasedDependencyInjectionExample(Tyre tyre, HeadLights headlight) {
        this.tyre = tyre;
        this.headlight = headlight;
    }
}
class Tyre {
    public Tyre() { }
}
class HeadLights {
    public HeadLights() {}
}
```

Constructor argument resolution: As we have seen in the example in Section 20.1.2.3.1, we are passing two parameters to the constructor ConstructorBasedDependencyInjectionExample. Since these two parameters are of different types and not related by inheritance, there is no ambiguity and hence it is easier to resolve the matching process. In this case, the order in which the parameters are supplied will be used to instantiate the bean. Hence, the following configuration will work without a need to explicitly specify indexes and/or types in the *<constructor-arg/>*. Let us see the following example:

```
<beans>
<bean id="ConstructorBasedDependencyInjectionExample" class="java11.fundamentals.
chapter20.ConstructorBasedDependencyInjectionExample">
    <constructor-arg ref="tyre"/>
    <constructor-arg ref="headlight"/>
</bean>
<bean id="tyre" class="java11.fundamentals.chapter20.Tyre"/>
<bean id="headlight" class="java11.fundamentals.chapter20.HeadLights"/>
</beans>
```

Constructor argument type matching: In the above case, we are using beans as parameters, hence the type is known. This helps in matching the parameters easily. But what if we use simple types like <value>false</value>? It is not possible for Spring to figure out by itself whether it is a Boolean or String. Let us see the following case:

```
package java11.fundamentals.chapter20;
public class ConstructorBasedDependencyInjectionSimpleTypeExample {
    private Boolean healthy;
    private String name;
    public ConstructorBasedDependencyInjectionSimpleTypeExample(String name, Boolean
healthy) {
        this.name = name;
        this.healthy = healthy;
    }
}
```

In the above case, we need to explicitly mention the type in the configuration in order for Spring to resolve the arguments.

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="java11.
fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg type="java.lang.Boolean" value="true"/>
    <constructor-arg type="java.lang.String" value="Mayur Ramgir"/>
</bean>
</beans>
```

Constructor argument index: In case there are multiple arguments with same type for example two `int` type arguments, we can use index attribute to specify the value for the desired argument. This will resolve the ambiguity in specifying values to the arguments.

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="java11.
fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg index="0" value="true"/>
    <constructor-arg index="1" value="Mayur Ramgir"/>
</bean>
</beans>
```

Constructor argument name: You may also use name attribute to specify value to the required argument. See the following example:

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="java11.
fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg name="healthy" value="true"/>
    <constructor-arg name="name" value="Mayur Ramgir"/>
</bean>
</beans>
```

20.1.2.3.2 Setter-Based Dependency Injection

In setter-based DI, Spring container instantiates the bean by invoking a no-argument constructor or no-argument static factory method. Once the bean is instantiated, the container then uses setter methods to inject the dependencies.

Let us see the following example to understand this better.

```
package java11.fundamentals.chapter20;
public class SetterBasedDependencyInjectionExample {
    private Camera camera;
    public SetterBasedDependencyInjectionExample() {}
    public Camera getCamera() {
        return camera;
    }
    public void setCamera(Camera camera) {
        this.camera = camera;
    }
}
package java11.fundamentals.chapter20;
public class Camera {
    public Camera() {

    }
}
```

Following is the XML configuration specify the Camera dependency:

```
<bean id = "setterBasedDependencyInjectionExample" class = "java11.fundamentals.
chapter20.SetterBasedDependencyInjectionExample">
    <property name = "camera" ref = "camera"/>
</bean>
<bean id = "camera" class = "java11.fundamentals.chapter20.Camera"></bean>
```

As you can see, we need to use <property> to set the dependent bean reference. Please note that the "name" attribute must match with the field name and "ref" attribute must match the "id" attribute of the dependent bean reference.

Differentiate between setter injection and constructor injection.

20.1.2.3.3 Field-Based Dependency Injection

In field-based DI, the Spring container injects the dependencies on fields which are annotated as @Autowired once the class is instantiated. See the following example:

```
package java11.fundamentals.chapter20;
public class FieldBasedDependencyInjectionExample {

    @Autowired
    private Camera camera;
    public FieldBasedDependencyInjectionExample() {}
}
```

In the above example, Spring will inject the *Camera* bean since it is annotated with **@Autowired**.

This looks the simplest of all, but it is highly discouraged to use because of several drawbacks, which are as follows:

1. It is expensive to use field-based DI in place of constructor or setter as Spring uses reflection to inject the annotated dependencies.
2. It does not support final or immutable declared fields, as they will get instantiated at the time of class instantiation. Only constructor-based DI supports immutable dependencies.
3. Possibility of violating the single responsibility principle as in field-based DI for a class. It is not difficult at all to end up having a lot of dependencies without even realizing that the class now focuses on multiple functionalities and violates its single responsibility principle. This problem is clearly visible in case of constructor-based injection, as the large number of parameters in the constructor will be clearly visible and give the signal of violation.
4. It creates tightly coupled code, as the main reason of using this injection is to avoid defining getters and setters and/or constructor. This leads to a scenario where only Spring container can set these fields via reflection. This is because there are no getters or setters, so the dependencies will not be set outside the Spring container, making it dependent on using the Spring container to use reflection to set the dependencies on autowired fields.
5. It creates hidden dependencies, as these fields do not have public interfaces like getters and setters, nor are they exposed via constructors. Hence, the outside world is unaware of these dependencies.

20.1.2.4 Lazy Initialized Beans

This is an interesting functionality that allows initializing a bean on first request and not on the application startup. It helps in increasing application performance, as many times, you may not need all the beans loaded on the startup. This way it enhances initialization time. However, there is a danger of discovering configuration errors later on bean request. This may interrupt the running application.

Following is an example of specifying lazy initialization:

```
<bean id = "camera" class = "java11.fundamentals.chapter20.Camera" lazy-init = "true">
</bean>
```

Now that you know both IoC and DI concepts, you can see how easy it is to use the DI approach to feed the required dependencies to the bean. It greatly simplifies the application development and helps in resource management, which is crucial for performance driven applications. There is one more concept that you should know before we discuss Spring MVC. This concept is called aspect-oriented programming.

20.1.3 Aspect-Oriented Programming

Modularity is one of the important design principals to accomplish scalability and reusability. This can be achieved by subdividing the system into smaller parts keeping business logic separate from each other and focus on one particular functional area.

In the aspect-oriented programming (AOP) world, these standalone distinct parts are called *concerns*. Some of the functions which are distinct from the application's business logic like security, auditing, logging, caching, transactions, etc., may cross multiple areas of an application and are termed as *cross-cutting concerns*. AOP greatly helps along with DI, as DI focuses on decoupling objects from each other and AOP helps in decoupling cross-cutting concerns from the objects. We can also

relate AOP to *triggers* in programming languages like Java, .Net, Perl, etc. Similar to the trigger concept, Spring AOP offers *interceptors* which intercepts an application by inserting additional functionality on method execution such as before the method or after the method, without adding the required logic in the method itself.

QUICK CHALLENGE Explain the different AOP concepts and terminologies.

Following are some of the important concepts we should know about AOP.

1. **Aspect:** The standalone distinct modules are called *aspect*. For example, logging aspect, auditing aspect, etc.
2. **Join point:** As the name suggests, this is the point where aspects can be injected to perform a specific task.
3. **Advice:** It is the actual code block, which will be executed before or after the method execution.
4. **Pointcut:** It is a set of one or more join points where the actual code will get executed. It is a predicate which matches join points to execute advice. Pointcuts are specified using expressions or patterns. See the following example:

```
@Aspect
public class LoggingAspectPointcutExample {
    @PointCut("execution(* java11.fundamentals.*.*(..))")
    private void logData(){}
}
```

In the above example, we have used a few annotations. Let us understand their meanings.

1. **@Aspect:** This annotation specifies that the class contains advice methods.
2. **@PointCut:** This annotation specifies the JoinPoint where an advice should be executed.
3. **execution(* java11.fundamentals.*.*(..)):** This specifies on which methods the given advice should be applied.
4. **Introduction:** This enables the inclusion of new methods, fields or attributes to the existing classes. It is also known as inter-type declarations in *AspectJ*. In other words, with introduction, an aspect can assert advised objects to implement a specific interface. It can also provide implementation of this interface.
5. **Target object:** It specifies the object that is being advised by one or more aspects. This object will always be a *proxied* object as Spring AOP is implemented using runtime proxies.
6. **AOP proxy:** It denotes the object to implement the aspect contracts.
7. **Weaving:** It is a process to create an advised object by linking one or more aspects with other objects or application types. There are various ways it can be done such as compile, load, or at runtime. Spring AOP performs weaving at runtime.

Types of Advice

There are total of five advices that can be used in the applications. Each advice has a specific purpose to accomplish. Following are descriptions of these advices:

1. **before:** This advice type makes sure to run the stated advice before the method execution on which it is specified.
2. **after:** This advice type makes sure to run the stated advice after the method execution on which it is specified irrespective of this method's outcome.
3. **after-returning:** This advice type makes sure to run the stated advice only after the successful completion of the method on which it is specified.
4. **after-throwing:** This advice type makes sure to run the stated advice only on method exits by throwing an exception on which it is specified.
5. **around:** This advice type makes sure to run the stated advice before and after the method execution on which it is specified. This is type is more useful on the auditing and logging type of aspects.

QUICK CHALLENGE Explain Spring Aspect Oriented Programming (AOP) capabilities and goals.

20.2 | Spring Architecture

Now you have learned about IoC and DI, you must have comprehended that the Spring is built as a modular framework. Hence, you have an option to pick and choose the modules you need as per your requirement. Let us explore the modules offered by Spring Framework.

Overall, there are 20 modules that provide various functionalities. See Figure 20.2, which shows the Spring Framework's architecture diagram, to understand this better.

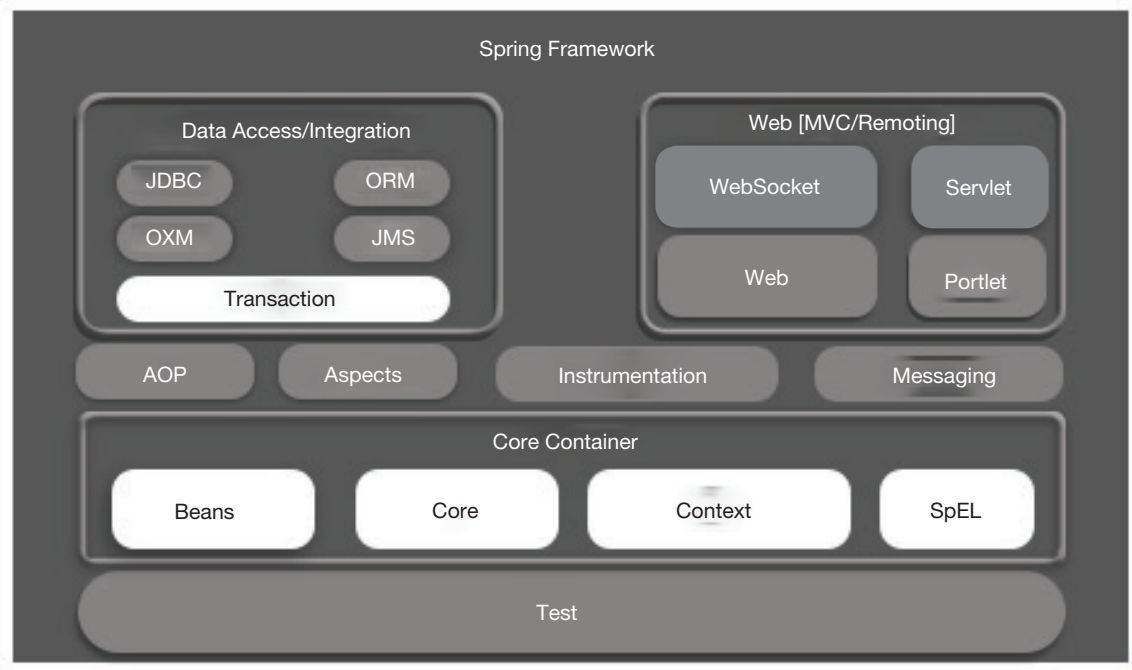

Figure 20.2 Spring framework architecture diagram.

20.2.1 Core Container

As you can see, the core container acts as the base of Spring Framework, which contains modules like Beans, Core, Context, and Expression Language (SpEL). All other modules rest on top of the core container. Let us study these modules in detail.

1. **Core module:** This is responsible for providing fundamental features like *IoC* and *DI,* which are the crucial parts of the framework that provide modularity.
2. **Bean module:** This provides implementation of a factory pattern known as *BeanFactory*.
3. **Context module:** This module is the provider of a well-known interface called *ApplicationContext*. The Context module uses the base offered by Core and Bean modules. *ApplicationContext* is the interface that provides configuration information to the application, which is necessary to instantiates the required beans. Spring builds up this interface upon starting the application and it is read-only at run-time but it can be reloaded if requires. Many classes implement this interface to provide various configuration options. *ApplicationContext* offers many bean factory methods, which can be used to access different application components. It provides various functionalities such as allowing publishing events to register listeners, loading file resources, supporting internationalization by resolving messages, etc.
4. **SpEL module:** This module provides a prevailing tool to query and manipulate an object graph at runtime. It supports various functionalities such as bean references, calling constructors, method invocation, collection projection, collection selection, relational operators, regular expressions, Boolean and relational operators, assignment, accessing properties, array construction, accessing arrays, accessing lists, accessing maps, literal expression, class expressions, user defined functions, inline maps, inline lists, variables, templated expressions, and ternary operator. It also offers various interfaces and classes such as ExpressionParser interface, EvaluationContext interface, Expression interface, SpelExpression class, SpelExpressionParser class, and StandardEvaluationContext.

Flash Quiz Does Spring provide transaction management?

20.2.2 Data Access/Integration

This section looks after accessing, storing, and manipulating data. This layer contains modules such as ORM, OXM, JDBC, JMS, and Transaction. Following are the descriptions of each module.

1. **ORM module:** ORM stands for object-relationship mapping, which offers an easy way to persist objects without writing database queries. It binds objects to the database tables, which hides the details of SQL queries from application logic. With ORM, objects can deal with other objects without worrying about underlying persistence mechanism. Spring does not offer ORM implementation but it provides a mechanism to integrate with external ORM tools like Hibernate, JPA, JDO, iBatis, etc. In the Chapter 21, we will explore Hibernate in detail.
2. **JDBC module:** This module provides an abstraction layer for JDBC, which is a Java API to interact with database to perform CRUD operations (CRUD stands for create, read, update, and delete). It uses JDBC drivers to connect with different providers databases.
3. **OXM module:** OXM stands for Object XML Mapping, which offers a mechanism to convert an object to an XML document and vice versa. XML is one of the important formats of transferring data to remote applications. With OXM, it is easier to convert the transmitted data into an object and prepare objects to transfer over network. The conversion from object to XML known as XML Marshalling or XML Serialization, and from XML to object known as Unmarshalling or Deserialization. This module is an extendible one and can integrate with other frameworks such as Castor, JAXB, and XStream.
4. **JMS module:** In distributed and modular systems, messaging plays a crucial role for communication between loosely coupled elements. Java provides an API called Java Message Service (JMS), which offers a mechanism to create, send, and read messages in a reliable and asynchronous way.
5. **Transaction management module:** Transaction management is a critical feature in RDBMS to promise data integrity and consistency. Database transactions are a series of actions that are considered as a single unit of work, which must complete totally or not at all. This module offers a reliable abstraction for transaction management which not only supports declarative transaction management but also offers a programmatic transaction management simpler than Java Transaction API (JTA). This module is consistent across different APIs such as JTA, Hibernate, Java Persistent API (JPA), JDBC, and Java Data Objects (JDO).

QUICK CHALLENGE Explain the benefits of ORM.

20.2.3 Web Container

This section provides various frameworks that are useful for web related applications, which offers integration with other popular MVC frameworks such as Struts and JSF. It has various modules such as Web, Web-Servlet, Web-Sockets, and Web-Portlet. Let us explore in detail.

1. **Web module:** This module offers web-related functionalities such as multiple file uploads, integration to IoC using a servlet, and web application context.
2. **Web-servlet module:** This module is an implementation of Model–View–Controller (MVC) pattern. It offers a clean separation between domain model code and web interface.
3. **Web-socket module:** This module provides support of Web-Socket, which is a bidirectional communication between server and client.
4. **Web-portlet module:** This module provides similar functionality as Web-Socket module for implementing MVC in a portlet environment.

Flash Quiz What is the role of web container?

20.2.4 Other Modules

1. **AOP module:** This module provides Aspect Oriented Programming support to Spring Framework. As we have learned in Section 21.1.3, AOP provides a simple mechanism to implement cross-cutting concerns.
2. **Aspects module:** This module supports integration with AspectJ.
3. **Testing module:** This module provides provision to implement integration and unit tests. With the help of this module, JUnit or TestNG type of frameworks can be used in Spring Framework.
4. **Instrumentation module:** This module provides class instrument support and classloader implementations, which are useful in various application servers.
5. **Messaging module:** This module provides foundation for messaging-based applications, which contains a set of annotations for mapping messages to methods.

Due to the modularity of the Spring Framework, you are free to choose the modules you need without breaking any core framework's functionality.

20.3 | Spring MVC

Spring MVC is one of the most popular Java website frameworks to program website applications. Similar to other industry technologies, it makes use of the model view design. All the general Spring features such as DI and IoC are implemented by Spring MVC.

To use the Spring framework, Spring MVC makes use of class DispatcherServlet. This class processes an incoming request and assigns it with models, controller, views, or any of the right resource.

If you are unfamiliar with MVC pattern, then bear in mind that it is a software architectural pattern. The data of the application is referred to as the model which can be a single or multiple objects. For the business logic, a controller is used which acts as a supervisor. The end user is provided a perspective through a "view".

In Spring MVC, any class which uses the annotation "@Controller" is the controller of the application. For views, a combination of JSP and JSTL can be utilized, though developers can also use other technologies. For the front controller, the DispatcherServlet class is used in Spring MVC.

When a request arrives, it is discovered by the DispatcherServlet class. This class gets the required information from the XML files and delivers the request to the controller. Subsequently, the controller generates a ModelAndView object. Afterwards, the DispatcherServlet class processes the view resolver and calls the appropriate view.

20.3.1 DispatcherServlet

The DispatcherServlet is used for request processing. It has to be mapped and declared in accordance with the Servlet specification. This servlet utilizes configurations in Spring to get information related to the delegate components for exception handling, view resolution, and request mapping.

20.3.1.1 Context Hierarchy

For configuration purposes, the DispatcherServlet requires the WebApplicationContext. It is linked with the Servlet and ServletContext. It is also attached to the ServletContext so the methods which are associated with it can utilize the RequestContextUtils.

For most of the applications, one WebApplicationContext is enough. However, sometimes there is a context hierarchy in which a single root WebApplicationContext is used for sharing between more than one instance of DispatcherServlet. Each of them has its configuration of own child WebApplicationContext.

Generally, the root WebApplicationContext stores data repositories, business services, and other infrastructure beans. These are shared between the Servlet instances. It is possible to override these beans as they are inherited via the WebApplicationContext of the Servlet children.

Special Bean Types

1. **HandlerMapping:** It is used with a handler for mapping requests. It uses the HandlerMapping implementation.
2. **HandlerAdapter:** It uses the DispatcherServlet for invoking handlers which are mapped with a request.

3. **HandlerException resolver:** It is used for resolving exceptions where they are mapped with handlers and error views of the HTML.
4. **View resolver:** It is used to resolve the view names which rely on logical Strings.
5. **Locale resolver:** It is used for resolving a client's locale or timezone.
6. **Themes resolver:** It is used to resolve the themes of a web application.
7. **MultiplepartResolver:** It is used for abstraction in order to help with parsing a request which has multiple parts.
8. **FlashMapManager:** It is used for storing and retrieving the input/output FlashMap.

20.3.2 Spring MVC Processing

The DispatcherServlet is used for the following:

1. It searches the WebApplicationContext and attaches the request in the form of an attribute so other components such as controller can use it.
2. The request is bound to the locale resolver which allows the process' elements in resolving the locale while processing and preparing data, rendering the view, or other requests.
3. Requests are bound to the theme resolver which allows views and other elements to use the theme.
4. When a multipart file resolver is defined, all the parts of the request are processed. After multiparts are discovered, the MultipartHttpServletRequest is wrapped to help with processing.
5. When the right handler is found, controllers, preprocessors, postprocessors, and others in the handler's executive chain are executed so the model is prepared or rendered. For annotated controllers, it is possible to render the response.
6. The view is rendered when a model is returned. When it is not returned, there is no need for view rendering as the request is already processed.
7. The declared beans in HandlerExceptionResolver resolve exceptions which come up in the request processing.

The Spring DispatcherServlet helps in returning the "last-modification-date", which is defined in the Servlet API. In order to assess this date, a simple processing is used.

1. The DispatcherServlet searches for the right handler mapping.
2. The DispatcherServlet then checks if the handler has implemented the "LastModified" interface.
3. If verified, the "long getLastModified" method value of the interface is sent to the client.

Can you configure Spring application with Configuration file and Annotations?

20.4 | Interception

In all of the Spring MVC HandlerMapping implementations, interceptors are supported. They are used to implement certain functionality for specific requests. For instance, they can be used to check for a principal. It is necessary for an interceptor to implement "HandlerInterceptor", which is included in the org.springframework.webservlet having the following methods for preprocessing and post-processing.

1. `preHandle():` It is used for the time period before the handler is executed.
2. `postHandle():` It is used when the handler has been executed.
3. `afterCompletion():` It is used after the request has been completed the execution.

20.5 | Chain of Resolvers

When you declare multiple beans of HandlerExcpetionResolver, you can create an exception resolver chain in your SpringMVC application where you have to configure and define the order properties accordingly. If the order property is high, then the exception resolver is processed accordingly. With respect to the HandlerExceptionResolver, it returns the following.

1. An empty ModelAndView in case the resolver handled the exception.
2. A Model and View which refers to an error view.
3. In case the exception is not resolved, it returns "null" for other resolvers. On the other hand, if the exception is still staying till the end, the Servlet container can be used to bubble it up.

Can you manage concurrent sessions using Spring MVC? If so, what are the things you need to consider?

20.6 | View Resolution

The View and ViewResolver interfaces are used by the SpringMVC, which helps in rendering models present in a browser while avoiding dependence on a certain technology for viewing. The ViewResolver is used to map actual views and view names. The View manages the preparation data before it is passed over to a certain technology for viewing.

It is possible to use multiple resolver beans to chain the view resolvers. Sometimes, the order property is used for defining ordering with it. To configure the view resolution, you just have to add the ViewResolver beans in the configuration of your SpringMVC application.

A Basic Example

Consider the following example for a basic Spring Web MVC project. Use Maven and add dependency in the pom.xml file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.introspring</groupId>
  <artifactId>SpringMVC</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVC Example</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
  </dependencies>
  <build>
    <finalName>SpringMVC</finalName>
  </build>
</project>
```

Now generate a class for the controller and add the following two annotations. The @Requestmapping annotation is required for mapping the appropriate URL name with the class. The @Controller class is simply used to treat a class as the controller.

```
package com.introspring;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass {
@RequestMapping("/")
    public String show()
    {
        return "index";
    }
}
```

In the XML file, make sure to write about the DispatcherServlet class which is serves as the front controller.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>Model View Controller</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

In the spring-servlet.xml file, define the components for the views. This XML file must be placed in the WEB-INF directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Specify the functionality to scan components -->
    <context:component-scan base-package="com.introspring" />

    <!—Specify the functionality for formatting, validation, and conversion -->
    <mvc:annotation-driven/>
</beans>
```

Generate an index JSP page and write the following.

```
<html>
<body>
<p>We are learning Spring MVC</p>
</body>
</html>
```

As an output, you can see "We are learning Spring MVC" text on your page. This message was delivered to you through the logic of the Controller.

20.7 | Multiple View Pages

In this example, we would apply redirection between two view pages. Begin by adding dependencies in the pom.xml file.

```
 <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Afterward, you have to generate a page for the request. To do this, create a JSP page and add a hyperlink in it.

```
<html>
<body>
<a href="Success">Hit This Link...</a>
</body>
</html>
```

Now generate a class for the controller which can process the JSP pages and return them. For mapping of the class, use the @Requestmapping annotation. We would add our "href" value in the following code so our Controller can easily manage and view our HTML elements.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass {
@RequestMapping("/success")
    public String reassign()
    {
        return "view";
    }
@RequestMapping("/success again")
public String show()
{
    return "final";
}
}
```

In the web.xml file, place a controller entry.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Now, define the bean for XML file. You have to add the view resolver in the component of the view. For the ViewResolver, the InternalResourceViewResolver class can be used. The XML file must be placed in the WEB-INF directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!—Add functionality to scan components -->
    <context:component-scan base-package="com.ith" />

    <!—Add functionality for formatting, validation, and conversion -->
    <mvc:annotation-driven/>
<!—Specify the view resolver for the Spring MVC -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceVie-
wResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

Now generate the components for the view. In the view.jsp, write the following.

```
<html>
<body>
<a href="successagain">Spring Tutorial</a>
</body>
</html>
```

In the last.jsp, write the following.

```
<html>
<body>
<p>Hello User, Let's Learn Spring MVC to Master the Java Backend</p>
</body>
</html>
```

When you run this application, you would first get a web page with a hyperlink. When you will click on it, it sends a request which is managed by the controller to reply with a response in the form of another page. Hit the link on the new page and you would be again redirected to another page, which would show you a welcome message. In this way, we have successfully gone over multiple web pages in Spring MVC.

20.8 | Multiple Controllers

Till now, we have been using a single controller in our examples. However, it is possible to generate multiple controllers in Spring MVC at the same time. Each of the controller class must be mapped properly with the @Controller annotation.

Begin by adding the dependencies in the pom.xml file as we have done before.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Then generate your initial page which would be initially viewed by the user. We would add two different hyperlinks for each of our controllers.

```
<html>
<body>
<a href="hiuser1">First User</a> ||
<a href="hiuser2">Second User</a>
</body>
</html>
```

Generate two classes for controller which can process and return the specified view page. For the first class, write the following code.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class User1 {
@RequestMapping("/hiuser1")
    public String show()
    {
        return "vp1";
    }
}
```

For the second class, write a similar piece of code.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class User2 {
@RequestMapping("/hiuser2")
    public String show()
    {
        return "vp2";
    }
}
```

Place an entry for the controller in the web.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Then specify the bean in the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!—Specify the view resolver in Spring MVC -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

Generate the components for the view. In the first vp1.jsp, write the following.

```
<html>
<body>
<p> In this example we have used two controllers.</p>
</body>
</html>
```

In the second vp2.jsp page, write the following code.

```
<html>
<body>
<p> In this example we have used two view pages.</p>
</body>
</html>
```

Run this code. In the output, the user would be presented two links. When the user clicks on the first page, they would be redirected to a new page and displayed a text. Clicking on the second link will take the user to a different page with different text. This is exactly how it happens in real world browsing. Hence, whenever you are clicking on links on the websites, then behind the scenes, each link is configured to redirect to a different page by their respective controllers.

Can you add security on Spring controllers to avoid denial of service attack? Explain.

20.9 | Model Interface

In the Spring MVC ecosystem, the model serves as a container which can be used to store the application's data. This data can be anything such as a record from the DB, an object, or simply a String. The controller must have the Model interface. The HttpServletRequest object processes the data given by a user and sends it to this interface. Any view page can then access that piece of data from model's interface.

As an example, let us generate a login page which has a username and password, a common use case in the web world. For validation, we can define a certain value. Add the required dependencies in the pom.xml.

```
 <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Now generate the login page which would get the user's credentials. We will name it as index.jsp.

```
<html>
<body>
<form action="hiuser">
UserName : <input type="text" name="name"/> <br><br>
Password : <input type="text" name="pass"/> <br><br>
<input type="submit" name="submit">
</form>
</body>
</html>
```

Now, generate the controller class in which the HttpServletRequest would be responsible to intercept the data which is typed by the user on the HTML form. The Model interface saves the data for the request and sends it to the view page.

```
import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass{
    @RequestMapping("/hiuser")
    public String show(HttpServletRequest r,Model md)
    {
        //read the data from the form
        String name=r.getParameter("name");
        String pass=r.getParameter("pass");
        if(pass.equals("abc12"))
        {
            String message="Welcome "+ name;
            //add a message to the model
            md.addAttribute("message", message);
            return "vp";
        }
        else
        {
            String msg="We apologize Mr. "+ name+". This is the wrong password";
            md.addAttribute("message", message);
            return "ep";
        }
    }
}
```

Now, open the web.xml file and add the controller's entry.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Afterward, configure the bean by setting up the spring-servlet.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!—Specify the view resolver in Spring MVC -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

For the components of the view, add two .jsp pages in the WEB-INF/jsp directory. The first page vp.jsp should look like the following. See how we are using the backend code to display our code dynamically on user's view by adding the "${message}" parameter.

```
<html>
<body>
${message}
</body>
</html>
```

The second page ep.jsp should show something like the following.

```
<html>
<body>
${message}
<br><br>
<jsp:include page="/index.jsp"></jsp:include>
</body>
</html>
```

When the user gets the username page, then there are two possible scenarios.

1. If the user types the correct password "abc12" and hits the submit button, then the controller would take them into a new page with a greeting.
2. On the other hand, a wrong password means that they would remain in the same page with a warning.

The basic idea behind the "login" functionality you see in websites is pretty much the same.

20.10 | RequestParam

The @RequestParam annotation reads data from the form and applies automatic binding for the method's available parameters. It does not consider the HttpServletRequest object for reading data. The annotation also applies mapping for the request parameter. If the type of the parameter in the method is "Map" along with the definition for the name of the

request parameter, then that parameter is changed into a Map. Otherwise, the name and values for the request parameter are placed in the map parameter.

As an example, let us generate a login page. Begin by placing the required dependencies in the pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Now, generate the page for the request which would take the credentials of the user. This page is saved as index.jsp.

```
<html>
<body>
<form action="hiuser">
Student Name : <input type="text" name="name"/> <br><br>
Password : <input type="text" name="pass"/> <br><br>
<input type="submit" name="submit">
</form>
</body>
</html>
```

Now, generate a controller class. In this class, the @RequestParam annotation reads the data from the form and applies binding for the request parameter.

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class ContClass {
    @RequestMapping("/hiuser")
    //read the provided form data
    public String display(@RequestParam("name") String name,@RequestParam
("pass") String pass,Model md)
    {
        if(pass.equals("abc12"))
        {
            String message="Welcome "+ name;
            //write a message for the model
            md.addAttribute("message", message);
            return "vp";
        }
        else
        {
            String msg="We apologize "+ name+". The typed password is wrong";
            md.addAttribute("message", message);
            return "ep";
        }
    }
}
```

For the vp page, write the following.

```
<html>
<body>
${message}
</body>
</html>
```

For the ep page, write the following.

```
<html>
<body>
${message}
<br><br>
<jsp:include page="/index.jsp"></jsp:include>
</body>
</html>
</html>
```

The output is similar to the previous example. However, the controller's working changed. In this example, we did not use the HttpServletRequest class. Instead, we used our @RequestParam annotation for our HTML elements. Through the annotation, we were able to apply automatic binding on HTML elements.

20.11 | Form Tag Library

Form tags in Spring MVC are the basic components of web pages. These tags can be reused and reconfigured according to the requirements of the users. They are extremely helpful for JSP for the readability and maintainability factors.

The library for the form tags is in the spring-webmvc.jar. For turning on the library's support, some configuration is required. Place the following code in the start of the JSP web page.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

The container tag is a sort of parent tag which stores all of the library's tag. Such tag creates a form tag for HTML.

```
<form:form action="nextFormPath" modelAttribute=?abc?>
```

Following are one of the most common form tags in Spring MVC.

1. **form:form –** Used to store all of the other tags.
2. **form:input –** Creates the text field for the user input.
3. **form:radiobutton –** Creates the radio button which can be marked or unmarked by the user.
4. **form:checkbox –** Creates the checkboxes which can be checked or unchecked by the user.
5. **form:password –** Creates a field for user to type password.
6. **form:select –** Creates a drop-down list for the user to choose an option.
7. **form:textarea –** Creates a field for text which spans multiple lines.
8. **form:hidden –** Creates an input field which is hidden.

20.12 | Form Text Field

The form text field creates an HTML input tag through the use of the bound value. For instance, consider a form where users can type their details to make a reservation in a restaurant. This example mirrors the real world reservations and booking structure.

To begin with, add the required dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>

    <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
```

Generate a bean class now. This class can store variables and the getter setter methods for the input fields of the form.

```
public class Reservation {
    private String fName;
    private String lName;
    public Reservation() { }
    public String getFname() {
        return Fname;
    }
    public void setFname(String Fname) {
        this.Fname = Fname;
    }
    public String getLname() {
        return Lname;
    }
    public void setLname(String Lname) {
        this.Lname = Lname;
    }
}
```

Now add a Controller class.

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@RequestMapping("/reservation")
@Controller
public class ContClass {
       @RequestMapping("/bkfrm")
    public String bkfrm(Model md)
    {
       //generate an object for reservation
        Reservation rsv=new Reservation();
       //pass the object to the model
        md.addAttribute("reservation", rsv);
        return "rsv-page";
    }
    @RequestMapping("/subfrm")
    // @ModelAttribute applies binding for the data of the form to the object
    public String submitForm(@ModelAttribute("reservation") Reservation rsv)
    {
        return "confirmation-form";
    }
}
```

You can then add the controller's entry in the web.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

In the spring-servlet.xml file, place the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!—Enables the scanning of components -->
    <context:component-scan base-package="com.ith" />
    <!—Enables formatting, validation, and conversion -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

To generate the requested page, place the following code in the index.jsp file.

```
<!DOCTYPE html>
<html>
<head>
    <title>Reservation Form for the Restaurant</title>
</head>
<body>
<a href="reservation/bkfrm">Click this link to get a booking at the restaurant</a>
</body>
</html>
```

For the reservation page, place the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
<head>
    <title>Restaurant Form</title>
</head>
<h3>Reservation Form for Restaurant</h3>
<body>
    <form:form action="subfrm" modelAttribute="reservation">
        First name: <form:input path="Fname" />
        <br><br>
        Last name: <form:input path="Lname" />
        <br><br>
        <input type="submit" value="Submit" />
    </form:form>
</body>
</html>
```

In the confirmation page, place the following code.

```
<!DOCTYPE html>
<html>
<body>
<p>Your reservation is confirmed successfully. Please, re-check the details.</p>
First Name : ${reservation.Fname} <br>
Last Name : ${reservation.Lname}
</body>
</html>
```

In the output, a hyperlink will take a user to a reservation form for the restaurant. When the user completes typing the details and submits it, then they are displayed their information as part of the rechecking procedure.

20.13 | CRUD Example

If you are learning a web framework, then it is important to equip yourself with enough knowledge that you can make a basic CRUD application. Such functionality is one of the most common client requirements and therefore getting the hang of it can be extremely useful for your career. To begin with, generate a new table in your database titled "students". Configure and enter the following fields in it.

1. ID
2. Name
3. GPA
4. Department

Now come to Eclipse IDE and begin the management of your dependencies by adding the following in the pom.xml file.

```
 <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
```

Subsequently, you have to create your bean class "Student" which contains variables according to the columns of your database.

```
public class Student {
private int id;
private String name;
private float GPA;
private String department;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getGPA() {
        return GPA;
    }
    public void setGPA(float GPA) {
        this.GPA = GPA;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
}
```

For the controller class, write the following code.

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.ith.beans.Student;
import com.ith.dao.StudentDao;
@Controller
```

```
public class StudentController {
    @Autowired
    StudentDao sDao;//Use the XML file for dao injection
 /* To take input, it generates a form. You can also see the reserved request attribute
in the form of "command". This attribute shows the data pertaining to the object in the
form.
     */
    @RequestMapping("/studentform")
    public String displayform(Model md){
        md.addAttribute("command", new Student());
        return "studentform";
    }
    /* This is done in order to use the database for storing data. The model
object receives request data from the @ModelAttribute. It is also necessary to add
RequestMethod.Post method as by default, the request type is set to GET.
*/
    @RequestMapping(value="/store",method = RequestMethod.POST)
    public String store(@ModelAttribute("student") Student student){
        sDao.store(student);
        return "redirect:/viewstudent";//
    }
    /* To show the student list which are stored in the model object */
    @RequestMapping("/viewstudent")
    public String viewstudent(Model md){
        List<Student> list=sDao.getStudents();
        md.addAttribute("list",list);
        return "viewstudent";
    }
    /* It shows the data of object from the form in accordance with the provided id.
      To add data in the variable, we have used the @PathVariable.  */
    @RequestMapping(value="/editstudent/{id}")
    public String edit(@PathVariable int id, Model md){
        Student student=sDao.getStudentById(id);
        md.addAttribute("command",student);
        return "studentupdateform";
    }
    /* It edits the object in the model. */
    @RequestMapping(value="/editsave",method = RequestMethod.POST)
    public String editsave(@ModelAttribute("student") Student student){
        sDao.update(student);
        return "redirect:/viewstudent";
    }
    /* It is used for the deletion of a record. */
    @RequestMapping(value="/deletestudent/{id}",method = RequestMethod.GET)
    public String delete(@PathVariable int id){
        sDao.delete(id);
        return "redirect:/viewstudent";
    }
}
```

Create a DAO class like the following.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.ith.beans.Student;

public class StudentDao {
JdbcTemplate tpe;

    public void setTemplate(JdbcTemplate tpe) {
        this.tpe = tpe;
    }
    public int save(Student s){
        String sql="insert into students (name,GPA,department) values('"+s.
getName()+"',"+s.getGPA()+",'"+p.getDepartment()+"')";
        return tpe.update(sql);
    }
    public int update(Student s){
        String sql="update students set name='"+p.getName()+"', GPA="+s.
getGPA()+",department='"+p.getDepartment()+"' where id="+s.getId()+"";
        return tpe.update(sql);
    }
    public int delete(int id){
        String sql="delete from students where id="+id+"";
        return tpe.update(sql);
    }
    public Student getStudentById(int id){
        String sql="select * from students where id=?";
        return tpe.queryForObject(sql, new Object[]
{id},new BeanPropertyRowMapper<Student>(Student.class));
    }
    public List<Student> getStudents(){
        return tpe.query("select * from students",new RowMapper<Student>(){
            public Student mapRow(ResultSet rs, int row) throws SQLException {
                Student s=new Student();
                s.setId(rs.getInt(1));
                s.setName(rs.getString(2));
                s.setGPA(rs.getFloat(3));
                s.setDepartment(rs.getString(4));
                return s;
            }
        });
    }
}
```

In the web.xml file, add the controller.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

In the XML file, specify the bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
<context:component-scan base-package="com.ith.controllers"></context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
<property name="url" value="jdbc:mysql://localhost:3306/test"></property>
<property name="username" value=""></property>
<property name="password" value=""></property>
</bean>

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="ds"></property>
</bean>

<bean id="sDao" class="com.ith.dao.StudentDao">
<property name="template" ref="jt"></property>
</bean>
</beans>
Generate a requested HTML page.
<a href="studentform">Add Students</a>
<a href="viewstudent">View Students</a>
In the JSP student form, add the following.
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

        <h1>Add New Student</h1>
       <form:form method="post" action="save">
        <table >
         <tr>
          <td> StudentName : </td>
          <td><form:input path="name"  /></td>
         </tr>
         <tr>
          <td>GPA :</td>
          <td><form:input path="GPA" /></td>
         </tr>
         <tr>
          <td>Department :</td>
          <td><form:input path="department" /></td>
         </tr>
         <tr>
          <td> </td>
          <td><input type="submit" value="Save" /></td>
         </tr>
        </table>
       </form:form>
```

By changing the project name in the following file, add the following code in the studenteditform.jsp file.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

        <h1>Edit Student</h1>
       <form:form method="POST" action="/SpringCRUDExample/editsave">
        <table >
        <tr>
        <td></td>
         <td><form:hidden  path="id" /></td>
         </tr>
         <tr>
          <td>Name : </td>
          <td><form:input path="name"  /></td>
         </tr>
         <tr>
          <td>GPA :</td>
          <td><form:input path="GPA" /></td>
         </tr>
         <tr>
          <td>Department :</td>
          <td><form:input path="department" /></td>
         </tr>

         <tr>
          <td> </td>
          <td><input type="submit" value="Edit Save" /></td>
         </tr>
        </table>
       </form:form>
```

In the viewstudent file, add the following code.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
   <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Student List</h1>
<table border="2" width="70%" cellpadding="2">
<tr><th>Id</th><th>StudentName</th><th>GPA</th><th>Department</th><th>Edit</th><th>
Delete</th></tr>
   <c:forEach var="student" items="${list}">
   <tr>
   <td>${student.id}</td>
   <td>${student.name}</td>
   <td>${student.GPA}</td>
   <td>${student.department}</td>
   <td><a href="editstudent/${student.id}">Edit</a></td>
   <td><a href="deletestudent/${student.id}">Delete</a></td>
   </tr>
   </c:forEach>
   </table>
   <br/>
   <a href="studentform">Add New Student</a>
```

When this application is run, the user gets two links; they can either add a student or see the student table. Since this is the first time you are using this application, first focus on adding the entries. Click the add button, and then you are provided with

a form to type the required details. When these details are completed, then it can be saved and the data would then be stored in the database. A user can then use this step multiple times to populate the table. After generating a table, you can modify or delete the data in the table. Click on either one of them and perform your required task. If you choose "Edit" then you can revisit the form and modify any detail. On the other hand, if you click on "Delete" then you can permanently remove the record from your database. The database would update accordingly.

You can use the logic implemented in this application in various other applications. For instance, instead of creating a student list, you can modify the details and make it a grocery shopping list where you can always adjust your items.

20.14 | File Upload in Spring MVC

In Spring MVC, you can easily incorporate the functionality to upload files in various formats. Consider the following example.

At the start, you have to download and load the Spring Core, Spring Web, commons-fileupload.jar, and commons-io.jar file.

Afterward, place the commons-io and fileupload.jar files. In the spring-servlet.xml file, write the following:

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
Generate a form for the submission of file.
<form action="savefile" method="post" enctype="multipart/form-data">
Choose a File: <input type="file" name="file"/>
<input type="submit" value="Upload the File"/>
</form>
```

For the controller,

```
@RequestMapping(value="/savefile",method=RequestMethod.POST)
public ModelAndView upload(@RequestParam CommonsMultipartFile file,HttpSession session){
        String pth=session.getServletContext().getRealPath("/");
        String fname=file.getOriginalFilename();
        System.out.println(pth+" "+fname);
        try{
          byte b[]=file.getBytes();

          BufferedOutputStream bos=new BufferedOutputStream(
                 new FileOutputStream(path+"/"+fname));
          bos.write(b);
          bos.flush();
          bos.close();
        }catch(Exception e){System.out.println(e);}
        return new ModelAndView("upload-success","fname",pth+"/"+fname);
    }
```

To show your image in JSP, add the following code.

```
<h1>The Upload Is Successful.</h1>
<img src="${fname}"/>
To store the files in your project, generate a directory, "images".
<a href="uploadform">Upload the Image</a>
```

In the Student class, write the following:

```
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.commons.CommonsMultipartFile;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {
    private static final String UPLOAD_DIRECTORY ="/images";

    @RequestMapping("uploadform")
    public ModelAndView uploadForm(){
        return new ModelAndView("uploadform");
    }

    @RequestMapping(value="savefile",method=RequestMethod.POST)
    public ModelAndView saveimage( @RequestParam CommonsMultipartFile file,
           HttpSession session) throws Exception{
    ServletContext context = session.getServletContext();
    String pth = context.getRealPath(UPLOAD_DIRECTORY);
    String fname = file.getOriginalFilename();
    System.out.println(pth+" "+fname);
    byte[] b = file.getBytes();
    BufferedOutputStream str =new BufferedOutputStream(new FileOutputStream(
         new File(pth + File.separator + fname)));
    str.write(b);
    str.flush();
    str.close();
    return new ModelAndView("uploadform","filesuccess","The file is saved successfully!");
    }
}
```

In the web.xml file, write the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
 <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Subsequently, build a bean and type the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.ith"></context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

</beans>
```

The action for the form must be "post" in the following format.

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<!DOCTYPE html>
<html>
 <head>
 <title>Upload the Image</title>
 </head>
 <body>
<h1>File Upload Example in Spring MVC</h1>

<h3 style="color:blue">${filesuccess}</h3>
<form:form method="post" action="savefile" enctype="multipart/form-data">
<p><label for="image">Pick any image</label></p>
<p><input name="file" id="fileToUpload" type="file" /></p>
<p><input type="submit" value="Upload"></p>
</form:form>
</body>
</html>
```

In the output, the user first got a hyperlink. After clicking and redirecting into a new web page, the user was able to find a screen with a file upload option. When you upload a file or an image through this button, it would not show in the browser (or the client side). Instead, the file or image is sent to the server. Have your path printed via the console of the server and check the file.

20.15 | Validation in Spring MVC

Validation in Spring MVC is required for the restriction of user input so only the authorized users can proceed through a website.

In order to execute this validation, the Bean Validation API is used by the Spring MVC. This validation can be used for both the client-side and server-side programming. To use annotations and implement restrictions on object model, we use the Bean Validation API. By validation, we can validate a regular expression, a number, length, and other related input. It is also possible to offer some custom validations.

This API requires implementation because it is a type of specification. Hence, it makes use of the Hibernate Validator. Usually, the following annotations are a recurring theme in validation.

1. **@Min –** It calculates and ensures that a given number is equal to or greater than a provided value.
2. **@Max –** It calculates and ensures that a given number is equal to or less than a provided value.
3. **@NotNull –** It calculates and ensures that no null value is possible for a field.
4. **@Pattern –** It calculates and ensures that the provided regular expression is followed by the sequence.

Let us work on an example which contains a form with basic input fields. We would use the "*" sign to represent mandatory fields that must be filled in or else the form causes an error. Begin by adding the required dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
Now, generate a bean class like the following.
import javax.validation.constraints.Size;

public class Student {
    private String sname;
    @Size(min=1,message="required")
    private String pwd;

    public String getSname() {
        return sname;
    }
    public void setSname(String sname) {
        this.sname = sname;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

For your controller class, write the following code where the @Valid annotation implements rules for validation to the given object while the interface "BindingResult" stores the validation's output.

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hello")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/helloagain")
    public String subfrm( @Valid @ModelAttribute("stu") Student s, BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
        return "last";
        }
    }
}
```

In the web.xml file, adjust the following settings for the controller.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Now specify the bean,

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!—Add functionality to scan component -->
    <context:component-scan base-package="com.ith" />
    <!—Add functionality for formatting, conversion, and validation -->
    <mvc:annotation-driven/>
    <!—Specify the view resolver for Spring MVC -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

For the request page, generate an index.jsp file and place the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Hit This Link</a>
</body>
</html>
```

For the view components, generate the vp.jsp page with the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiuser" modelAttribute="stu">
Student Name: <form:input path="sname"/> <br><br>
Password(*): <form:password path="pwd"/>
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

And finally, add a last.jsp page:

```
<html>
<body>
Student Name: ${stu.sname} <br><br>
Password: ${stu.pwd}
</body>
</html>
```

Run this application. The user first finds a hyperlink which redirects to a web page. Afterward, the name and password details of the user are typed. However, this time the "password" field is displayed as "required" when you do not enter it. After typing the password, the screen shows your name along with your password.

20.16 | Validation with Regular Expression

In Spring MVC, we can use validation of user input with a specific technique, which is also known as *regular expression*. In order to utilize the validation for regular expression, you have to use the @Pattern annotation. In such a strategy, the regexp attribute is used with the required expression along with our annotation.

Add dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

Now make the Student.java class which is our bean class.

```
import javax.validation.constraints.Pattern;
public class Student {
    private String sname;
    @Pattern(regexp="^[a-zA-Z0-9]{4}",message="The length should be at least 4")
    private String pwd;

    public String getSname() {
        return sname;
    }
    public void setSname(String sname) {
        this.sname = sname;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

For the controller class, write the following code.

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hello")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/helloagain")
    public String subfrm(@Valid @ModelAttribute("stu") Student s BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
        return "last";
        }
    }
}
```

In the web.xml file, write the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Now, for the xml file, specify the bean like this in the spring-servlet file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!—Specify the view resolver in Spring MVC -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

Now, for the initial request, create an index.jsp page and add the following code.

```
index.jsp
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Hit This Link To Go Forward!</a>
</body>
</html>
```

To show the view components, create a vp.jsp page and add the following.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiuser" modelAttribute="stu">
Student Name: <form:input path="name"/> <br><br>
Password(*): <form:password path="pass"/>
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

Finally, create one more, last.jsp and add the following.

```
<html>
<body>
Username: ${stu.sname} <br><br>
Password: ${stu.pwd}
</body>
</html>
```

When the application is run, a click leads the user to a new screen which takes the username credential. If the password is shorter than 4 characters, then there is a warning text for it. On typing the right password, the application saves it and proceeds.

20.17 | Validation with Numbers

In Spring MVC, we can use validation of user input with a set of numbers. In order to utilize the validation for regular expression, you have to use @Min annotation and @Max annotation. The input has to be more than its value for the former, whereas the input has to lesser than its value for the latter.

Add dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
```

```
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
 <!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

Now make the Student.java class which is our bean class.

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.Size;

public class Student {
    private String name;
    @Size(min=1,message="required")
    private String pwd;

    @Min(value=50, message="A student must score 50 or more to qualify.")
    @Max(value=100, message="A student cannot score equal or less than 100.")
    private int marks;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
    public int getMarks() {
        return marks;
    }
    public void setMarks(int marks) {
        this.marks = marks;
    }
}
```

For your controller class, type the following code:

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hiuser")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/hiagain")
    public String subfrm( @Valid @ModelAttribute("stu") Student s, BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
        return "last";
        }
    }
}
```

In the web.xml file, add the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.
sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Specify a bean in the XML file. Then for the request page, add the following in the index.jsp.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hiuser">Hit this link!</a>
</body>
</html>
```

Now, to generate the view components, write the following:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiagain" modelAttribute="student">
Name: <form:input path="name"/> <br><br>
Password: <form:password path="pass"/>
<form:errors path="pass" cssClass="error"/><br><br>
Marks: <form:input path="marks"/>
<form:errors path="marks" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

In the last.jsp, type the following code:

```
<html>
<body>
Username: ${param.name} <br>
Password: ${param.pwd} <br>
Age: ${param.marks } <br>
</body>
</html>
```

This code would ensure that only students with marks in the range of 50–100 can submit their details.

Summary

Spring MVC remains one of the leading Java frameworks in the industry today. If you plan to learn web development for educational purposes or to begin a career, then it is an excellent choice. From the government sector to the banking sector, Java Spring MVC is used in a wide range of industries, especially in the enterprises.

In this chapter, we have learned the following concepts:

1. Spring Framework and Spring MVC.
2. Dependency Injection, Inversion of Control, and aspect-oriented programming.
3. Spring architecture including core module, web module, Bean module, and context module.
4. Bean scope such as singleton, prototype, and session.
5. Spring MVC concepts.
6. DispatcherServlet, Context Hierarchy, Special Bean Types, etc.
7. Spring Bean Processing, Interception, Chain of Resolver.
8. View Resolution, Multiple Controller, etc.
9. Various examples to understand concepts such as Model Interface and RequestParam.

In Chapter 21, we will learn about a popular ORM tool known as Hibernate. We will learn how to use ORM tools such as annotations, inheritance mapping, and class to table mapping.

Multiple-Choice Questions

1. Which one of the following components intercepts all requests in a Spring MVC application?
   (a) DispatcherServlet
   (b) ControllerServlet
   (c) FilterDispatcher
   (d) None of the above
2. Which one of the following components is utilized to map a request to a method of a controller?
   (a) URL Mapper
   (b) RequestResolver
   (c) RequestMapper
   (d) RequestMapping
3. If we want to apply custom validation, which one of the following interfaces is required to be implemented?
   (a) Validatable
   (b) ValidationAware
   (c) Validator
   (d) None of the above
4. Is is possible to generate multiple controllers in Spring MVC at the same time?
   (a) Yes
   (b) No
5. _____ uses the DispatcherServlet for invoking handlers which are mapped with a request.
   (a) HandlerMapping
   (b) HandlerAdapter
   (c) LocalResolver
   (d) FlashMapManager

Review Questions

1. How do you define Spring Framework?
2. What are the core modules of Spring Framework?
3. What is Dependency Injection?
4. What are the benefits of using Dependency Injection?
5. What is Inversion of Control?
6. How does Spring use Inversion of Control? What are the benefits?
7. What is aspect-oriented programming? How do we use it?
8. What are Bean Scopes?
9. What is the default Bean Scope?
10. When do we use Prototype Scope?
11. What is Spring MVC?
12. What is Dispatcher Servlet?
13. How does Autowiring work?
14. How does constructor-based Dependency Injection work?
15. How does setter-based Dependency Injection work?
16. How does field-based Dependency Injection work?
17. Which type of Dependency Injection is better to use? Why?
18. What types of advice are there? Explain in detail.
19. What is Pointcut?
20. What is Join Point?
21. What is the difference between Pointcut and Join Point?
22. What is Target object?
23. What is AOP Proxy?
24. What modules are there in the Data Access/Integration layer?
25. What is the use of Messaging Module?
26. What is context hierarchy?
27. What is Chain of Resolvers? How does it work?
28. What is view resolution?
29. What is the difference between @RequestParam and @PathVariable?

Exercises

1. Setup an environment either using Eclipse or Spring Tool Suite and create a Spring MVC project. Create three views, first to capture student data for registration using FormTag Library, second to display data and third home page to show a school page.
2. Extend this first exercise to add validations on the registration form.

Project Idea

Create a Hotel Booking application using Spring MVC. This app should have frontend view to search for hotels based on users' criteria such as star ratings, reviews, and price. And a feature to sort results based price or ratings. It should also have frontend view for admin to add hotel information. Use HTML, CSS, JQuery, and Bootstrap for the front end and use Spring MVC on the back end side.

Recommended Readings

1. Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho. 2017. *Pro Spring 5: An In-Depth Guide to the Spring Framework and its Tools*. Apress: New York
2. Ranga Karanam. 2017. *Mastering Spring 5.0*. Packt: Birmingham
3. Tejaswini Mandar Jog. 2017. *Learning Spring 5.0: Build enterprise grade applications using Spring MVC, ORM Hibernate and RESTful APIs*. Packt: Birmingham